Collaborators: Yining Liu(yl2nr), Yuhao Niu(bhb9ba)

Sources:

PROBLEM 1 *Scenic Highways*

In this problem we'll describe something similar to solving a problem with Dijkstra's algorithm, and ask you some questions about this new problem and a possible algorithm.

You are taking a driving vacation from town to town until you reach your destination, and you don't care what route you take as long as you maximize the number of scenic highways between towns that are part of your route. You model this as a weighted graph G, where towns are nodes and there is an edge for each route between two towns. An edge has a weight of 1 if the route is a scenic highway and a 0 if it is not.

You need to find the path between two nodes s and t that maximizes the number of scenic highways included in the path. You design a greedy algorithm that builds a tree like Dijkstra's algorithm does, where one node is added to a tree at each step. From the nodes adjacent to the tree-nodes that have already been selected, your algorithm uses this greedy choice:

Choose the node that includes the most scenic highways on the path back to the start node s.

When node t is added to the tree, stop and report that path found from s to t. Reminder (in case it helps you): the greedy choice for Dijkstra's algorithm was:

Choose the node that has the shortest path back to the start node s.

1. What is the key-value stored for each item in the priority-queue? Also, what priority-queue operation must be used when we need to update a key for an item already stored in the priority-queue?

   Solution:

   1. The key value (pair) stored should be the town name (as a vertex) and the weight of the route (as an edge). All the key values should first have $-\infty$ as the value like in Dijkstra's algorithm.

   2. We need to use *IncreaseKey()* operation. It allows us to update the key value. Also we can get the newly sorted queue that has the max value at the first. *getMax()* is also needed to get the route's scenic highway value.

2. Draw one or more graphs to convince yourself that the greedy approach describes above does not work. Then explain in words as best you can the reason this approach fails or a situation that will cause it to fail (i.e., a counterexample).

   Solution:

   The algorithm fails when we apply the greedy approach when we check the subsection of the tree. The subsection might not be an optimal choice since there can be a next part of this substructure that produces an opposite result.

Assume we are at node S, and S connects to A and B.

A is connected to C only.

B is connected to D. Then D to E, and E to F.

If we use greedy selection and find out the weight(S, A) is 1 and weight(S, B) is 0, then by the greedy selection we will choose path (S, A).

But if weight(B, D), weight(D, E), and weight(E, F) are all 1, and that weight(A, C) is 0. The route selected previously based on weight(S, A) is then incorrect because there are more scenery highways if we choose (S, B). Since choosing B side will end in scenery highways of 3 and choosing A side will only have 1, the greedy selection fails in some cases and does not give the optimal result.

PROBLEM 2 *As You Wish*

Buttercup has given Westley a set of n tasks $T = t_1, \ldots, t_n$ to complete on the farm. Each task $t_i = (d_i, w_i)$ is associated with a deadline d_i and an estimated amount of time w_i needed to complete the task. To express his undying love to Buttercup, Westley strives to complete all the assigned tasks as early as possible. However, some deadlines might be a bit too demanding, so it may not be possible for him to finish a task by its deadline; some tasks may need extra time and therefore will be completed late. Your goal (inconceivable!) is to help Westley minimize the deadline overruns of any task; if he starts task t_i at time s_i, he will finish at time $f_i = s_i + w_i$. The deadline overrun (or lateness) of tasks—denoted L_i—for t_i is the value

$$L_i = \begin{cases} f_i - d_i & \text{if } f_i > d_i \\ 0 & \text{otherwise} \end{cases}$$

Design a polynomial-time algorithm that computes the optimal order W for Westley to complete Buttercup's tasks so as to minimize the maximum L_i across all tasks. That is, your algorithm should compute W that minimizes

$$\min_{W} \max_{i=1,\ldots,n} L_i$$

In other words, you do not want Westley to complete *any* task *too* late, so you minimize the deadline overrun of the task completed that is most past its deadline.

1. What is your algorithm's greedy choice property?

   Solution:

   We minimize the deadline overrun of the task completed that is most past its deadline. We prioritize tasks with earlier deadlines and finish them as soon as possible. At each time t_i, choose the task with the earliest deadline (least d) from all tasks that have a positive overrun ($L > 0$) to minimize the task overrun. If no task overruns, we start with the task with the least d so we do the coming due task. This is how I decided to sort, but the actual overrun time does not change even if we put those task that does not overrun first because the deadline and the time needed to complete a task are all fixed. The overall sequence from the sort should be non-decreasing based on deadlines.

2. What is the run-time of your algorithm?

   Solution:

   The time complexity is $O(n \log n)$ as each sorting takes $\log n$ times and the total sorting time is n times. Other tasks such as computing lateness is linear time.

3. Consider the following example list of tasks:

   $$T = \{(2,2), (11,1), (8,2), (1,5), (20,4), (4,3), (8,3)\}$$

   List the tasks in the order Westley should complete them. Then argue why there is no better result for the given tasks than the one your algorithm (and greedy choice property) found.

   Solution:

   Westley's order to complete the task: $(1,5); (2,2); (4,3); (8,3); (8,2); (11,1); (20,4)$

   Following the order of deadlines.

We can prove this by contradiction. Let's assume that there is another optimal sequence with two different pairs: (d_a, w_a) that **comes before** (d_b, w_b)

To be different from my sequence, there are only two possible scenarios. With current time s_0 and at the start of t_a.

Either 1. $s_0 + w_a - d_a \leq 0$ and $s_0 + w_b - d_b > 0$.

Or 2. Both overruns ($L > 0$) and $d_a > d_b$.

Then the new sequence will have an overrun of $L_{new} = max((s_0 + w_a - d_a),\ (s_0 + w_a + w_b - d_b))$. My original (old) overrun is $L_{old} = max((s_0 + w_b - d_b),\ (s_0 + w_b + w_a - d_a))$. The A and B are opposite. Based on the two scenarios listed above:

If it is case 1: $L_{new} = (s_0 + w_a + w_b - d_b) > L_{old} = (s_0 + w_b - d_b)$

If it is case 2: $L_{new} = (s_0 + w_a + w_b - d_b) > L_{old} = (s_0 + w_b + w_a - d_a)$ since $d_a > d_b$.

Hence, we show that the new sequence cannot be better than our original version as it has a larger result (L). The original sequence is the optimal one.

PROBLEM 3 *Course Scheduling*

The university registrar needs your help in assigning classrooms to courses for the spring semester. You are given a list of n courses, and for each course $1 \leq i \leq n$, you have its start time s_i and end time e_i. Give an $O(n \log n)$ algorithm that finds an assignment of courses to classrooms which minimizes the *total number* of classrooms required. Each classroom can be used for at most one course at any given time. Prove both the correctness and running time of your algorithm.

Solution:

1. We first sorted the given list of n courses based on ascending order of end time (morning courses at the beginning and evening courses at the end) so we know the exact time for a class to end to check for available classrooms. If some courses have the same end time, we sort them based on ascending order of start time (first start list first).

We hence create a list of classrooms that have the latest end time as a value in it. We first make the list empty.

Pop out the first course in the sorted list with the s_i, e_i. We check if there is an available classroom from the list for the course. That is to check if the classroom's end time is before the start time, meaning the classroom currently does not have a course when this upcoming course wants to use the classroom. If there is an available classroom on the list, we update the classroom's end time to be e_i. If there is no available classroom, we create a new classroom to the list and have the latest end time to be e_i, using another empty classroom to hold this course.

2. To prove the correctness of this algorithm, we can check from several aspects. First, we check that there are no overlapping courses. We use the sort method first to arrange the list into end-time ascending order to ensure that the later course pop from the list definitely has a later end time than what is poped previously. As we update our classroom list, the later-pop course will only have a later end time so the classroom end time only increase. Thus, the only possible overlap is that the start time is before the end time so the classroom is still in use. We avoid this by checking the start time to the classroom end time to ensure no such thing occurs. Overall, there is no overlapping in our method.

Second, we prove that we minimize the number of classrooms. Our algorithm also ensures this by detecting all listed classrooms' end times before adding a new classroom to the list. If there is a classroom empty for use, we will use that classroom first instead of adding another classroom to the list. Therefore, when we add a new classroom, it means we have already compared all existing classrooms, and none of them are available. Via our algorithm, it is impossible to decrease the number of classrooms in the list so our algorithm generates the minimized number of classrooms.

For the running time of the algorithm, we can apply merge or quick sort to sort the list of courses with $O(\log n)$. In the checking step, we can apply a binary heap to store the end time for each classroom, so to check for the end time (available classrooms) is also $O(\log n)$. Finally, there are n courses. The total time is $O(n \log n + \log n) \rightarrow O(n \log n)$.

PROBLEM 4 *Ubering in Arendelle*

After all of their adventures and in order to pick up some extra cash, Kristoff has decided to moonlight as the sole Uber driver in Arendelle with the help of his trusty reindeer Sven. They usually work after the large kingdom-wide festivities at the palace and take everyone home after the final dance. Unfortunately, since a reindeer can only carry one person at a time, they must take each guest home and then return to the palace to pick up the next guest.

There are n guests at the party, guests $1, 2, ..., n$. Since it's a small kingdom, Kristoff knows the destinations of each party guest, $d_1, d_2, ..., d_n$ respectively and he knows the distance to each guest's destination. He knows that it will take t_i time to take guest i home and return for the next guest. Some guests, however, are very generous and will leave bigger tips than others; let T_i be the tip Kristoff will receive from guest i when they are safely at home. Assume that guests are willing to wait after the party for Kristoff, and that Kristoff and Sven can take guests home in any order they want. Based on the order they choose to fulfill the Uber requests, let D_i be the time they return from dropping off guest i. Devise a greedy algorithm that helps Kristoff and Sven pick an Uber schedule that minimizes the quantity:

$$\sum_{i=1}^{n} T_i \cdot D_i.$$

In other words, they want to take the large tippers the fastest, but also want to take into consideration the travel time for each guest. Prove the correctness of your algorithm. (Hint: think about a property that is true about an optimal solution.)

Solution:

The T_i in this case is fixed for each customer, so to minimize $\sum_{i=1}^{n} T_i \cdot D_i$ is to find the lower D_i for larger T_i. Hence, the idea is to find the guest who gives more tips per trip (unit distance) and serve them first to get the tip, making the product of tips and time ($T_i \cdot D_i$) smaller.

1. Sort the list of guests according to the *descending order* of $\frac{T_i}{t_i}$. $D_i = t_i + t_{prevguest}$. When $t_{prevguest} = 0, D_i = t_i$.

2. With $t_{prevguest} = 0$ as the first guest, we pop out the first guest f from the sorted list and append the guest f to the Uber Schedule. Update the $t_{prevguest} = t_{prevguest} + t_f$.

3. Repeat the sorting and popping out (appending) step until the list is empty. Return the final Uber Schedule.

This algorithm is correct because in every substructure we select the one with the largest tip per distance (time), and by prioritizing this guest and minimizing time, we ensure the product of tips and time ($T_i \cdot D_i$) is minimized. Since every substructure is optimized and minimized, the final result that adds all substructure is minimized and optimal.

Thus, we use Exchange Argument, assuming there is an optimal sequence a that is different from ours. Consider at any time t, guest with T_x is the next to be served according to x. If we swap guest T_x with T_b which is the next guest scheduled based on our own sequence, the result differences: $T_b \cdot (t_{prevguest} + t_b) + T_x \cdot (t_{prevguest} + t_b + t_x) - T_x \cdot (t_{prevguest} + t_x) + T_b \cdot (t_{prevguest} + t_x + t_b) = T_x * t_b - T_b * t_x \leq 0$. Since we know $\frac{T_b}{t_b} \geq \frac{T_x}{t_x} \Rightarrow T_b * t_x \geq T_x * t_b$.

So, exchanging the guest with our own algorithm doesn't make the case worse, meaning our algorithm is correct.